Random Access to Grammar Compressed Strings

Philip Bille
phbi@imm.dtu.dk

Gad M. Landau
landau@cs.haifa.ac.il

Oren Weimann
oren.weimann@weizmann.ac.il

Abstract

Grammar based compression, where one replaces a long string by a small context-free grammar that generates the string, is a simple and powerful paradigm that captures many of the popular compression schemes, including the Lempel-Ziv family, Run-Length Encoding, Byte-Pair Encoding, Sequitur, and Re-Pair. In this paper, we present a novel grammar representation that allows efficient random access to any character or substring of S without decompressing S.

Let S be a string of length N compressed into a context-free grammar $\mathcal{S}$ of size n. We present two representations of $\mathcal{S}$ achieving either $O(n)$ construction time and space and $O(\log N \log\log N)$ random access time, or $O(n \cdot \alpha_k(n))$ construction time and space and $O(\log N)$ random access time. Here, $\alpha_k(n)$ is the inverse of the k^{th} row of Ackermann's function. Our representations extend to efficiently support decompression of any substring in S. Namely, we can decompress any substring of length m in the same complexity as a random access query and additional $O(m)$ time. Combining this with fast algorithms for standard uncompressed approximate string matching leads to several efficient algorithms for approximate string matching within grammar compressed strings without decompression. For instance, we can find all approximate occurrences of a pattern P with at most k errors in time $O(n(\min\{|P|k, k^4 + |P|\} + \log N) + \text{occ})$, where occ is the number of occurrences of P in S.

All of the above bounds significantly improve the currently best known results. To achieve these bounds, we introduce several new techniques and data structures of independent interest, including a predecessor data structure, a weighted ancestor data structure, and a compact representation of heavy-paths in grammars.

1 Introduction

Modern text databases, e.g. for biological and World Wide Web data, are huge. A typical way of storing them efficiently is to keep them in compressed form. The challenge arises when we want to access a small part of the data or search within it, e.g., if we want to retrieve a particular DNA sequence from a large collection of DNA sequences or search for approximate matches of a newly discovered DNA sequence. The naive way of achieving this would be to first decompress the entire data and then search within the uncompressed data. However, such decompression can be highly wasteful in terms of both time and space. Instead we want to support this functionality directly on the compressed data.

We focus on the following primitives. Let S be a string of length N given in a compressed representation $\mathcal{S}$ of size n. The *random access problem* is to compactly represent $\mathcal{S}$ while supporting fast random access queries, that is, given an index i, $1 \leq i \leq N$, report $S[i]$. More generally, we want to support *substring decompression*, that is, given a pair of indices i and j, $1 \leq i \leq j \leq N$, report the substring $S[i] \cdots S[j]$. The goal is to use little space for the representation of $\mathcal{S}$ while supporting fast random access and substring decompression. Once we obtain an efficient substring decompression method, it can then serve as a basis for a compressed version of classical pattern matching. Namely, given an (uncompressed) pattern string P and $\mathcal{S}$, the *compressed pattern matching problem* is to find all occurrences of P within S without decompressing $\mathcal{S}$. The goal here is to search more efficiently than to naively decompress $\mathcal{S}$ into S and then search for P in S. An important variant of the pattern matching problem is when we allow approximate matching (i.e., when P is allowed to appear in S with some errors).

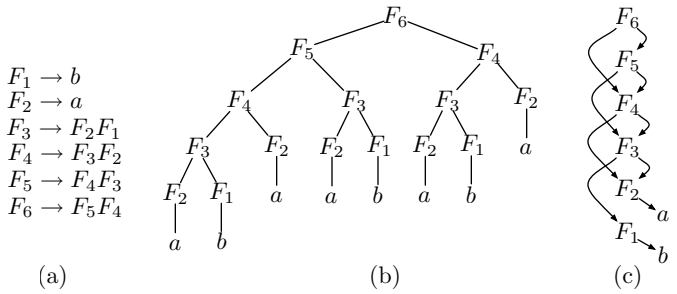


Figure 1: (a) A context-free grammar generating the string abaababa. (b) The corresponding parse tree. (c) The acyclic graph defined by the grammar.

We consider these problems in the context of *grammar-based compression*, where one replaces a long string by a small context-free grammar (CFG) that generates this string (see Fig. 1(a)). Such grammars that generate only one unique string capture many of the popular compression schemes including the Lempel-Ziv family [51, 54, 55], Sequitur [41], Run-Length Encoding, Byte-Pair Encoding [23, 46], Re-Pair [34], and many more [6–8,30,31,53]. All of these are or can be transformed into equivalent grammar-based compression schemes with no or little expansion [15,43]. In general, the size of the grammar, defined as the total number of symbol in all derivation rules, can be exponentially smaller than the string it generates.

Grammar based compression dates back to at least the 1970s [48, 49, 54, 55]. Since then, it has become popular in diverse areas outside of data compression including analysis of DNA [32, 40], music [41], natural language [20], and complexity analysis of sequences [15,43,54]. In the algorithmic perspective, the properties of compressed data were exploited for accelerating the solutions to classical problems on strings. In particular, compression was employed to accelerate both exact pattern matching [4,29,35,37,47] and approximate pattern matching [3, 9, 12–14, 19, 27–29, 36, 38].

Our Results

In this paper we present a new representation of CFGs that supports efficient random access and substring decompression. Let $\alpha_k(n)$ be the inverse of the k^{th} row of Ackermann's function[1]. We show the following.

Theorem 1 *For a CFG $\mathcal{S}$ of size n representing a string of length N we can support random access*

(i) in time $O(\log N \log\log N)$ after $O(n)$ preprocessing time and space, or

(ii) in time $O(\log N)$ after $O(n \cdot \alpha_k(n))$ preprocessing time and space for any fixed k.

Secondly, we consider the substring decompression problem. Clearly it is possible to decompress a substring of length m by doing m random access queries. We show that this can actually be done in the same complexity as a single random access query plus $O(m)$ time as stated by the following theorem.

Theorem 2 *For a CFG $\mathcal{S}$ of size n representing a string of length N we can decompress a substring of length m*

(i) in time $O(m + \log N \log\log N)$ after $O(n)$ preprocessing time and space, or

(ii) in time $O(m + \log N)$ after $O(n \cdot \alpha_k(n))$ preprocessing time and space for any fixed k.

[1]The inverse Ackermann function $\alpha_k(n)$ can be defined by $\alpha_k(n) = 1 + \alpha_k(\alpha_{k-1}(n))$ so that $\alpha_1(n) = n/2$, $\alpha_2(n) = \log n$, $\alpha_3(n) = \log^* n$, $\alpha_4(n) = \log^{**} n$ and so on. Here, $\log^{**} n$ is the number of times the $\log^*$ function is applied to n to produce a constant.

Finally, we show how to combine Theorem 2 with any black-box (uncompressed) approximate string matching algorithm to solve the corresponding compressed approximate string matching problem over grammar-compressed strings. We obtain the following connection between classical (uncompressed) and grammar compressed approximate string matching. Let $t(m)$ and $s(m)$ be the time and space bounds of some (uncompressed) approximate string matching algorithm on strings of lengths $O(m)$, and let occ be the number of occurrences of P in S.

Theorem 3 *Given a CFG $\mathcal{S}$ of size n representing a string of length N and a string P of length m we can find all approximate occurrences of P in $\mathcal{S}$*

(i) in time $O(n(m + t(m) + \log N \log\log N) + \text{occ})$ and space $O(n + m + s(m) + \text{occ})$, or

(ii) in time $O(n(m + t(m) + \log N) + \text{occ})$ and space $O(n \cdot \alpha_k(n) + m + s(m) + \text{occ})$.

We next describe how our work relates to existing results. We assume without loss of generality that the grammars are in fact *straight-line programs* (SLPs) and so on the righthand side of each grammar rule there are either exactly two variables or one terminal symbol. We further make the assumption that $\log N$ bits fit in a machine word. This is a fair assumption as the input i to a random access query is also of $\log N$ bits.

Related Work

The random access problem. It is easy to derive the following two simple trade-offs. If we use $O(N)$ space we can access any character in constant time by storing S explicitly in an array. Alternatively, if we are willing to settle for $O(n)$ random access time then the space can be reduced to $O(n)$ as well. To achieve this, we compute and store the sizes of strings derived by each grammar symbol in $\mathcal{S}$. This only requires $O(n)$ space and allows to simulate a top-down search expanding the grammar's derivation tree in constant time per node. Consequently, a random access takes time $O(h)$, where h is the height of the derivation tree and can be $\Omega(n)$. Surprisingly, the only known improvement to these trivial bounds is a recent succinct representation of grammars, due to Claude and Navarro [17]. They reduce the space from $O(n \log N)$ *bits* to $O(n \log n) + n \log N$ bits at the cost of increasing the query time to $O(h \log n)$.

The substring decompression problem. Using the simple random access trade-off we get an $O(n)$ space solution that supports substring decompression in $O(hm)$ time. Gasieniec et al. [24, 25] showed how to improve the decompression time to $O(h + m)$ while maintaining $O(n)$ space. Furthermore, the above mentioned succinct representation of Claude and Navarro [17] supports substring decompression in time $O((m + h) \log n)$.

The compressed pattern matching problem. Recall that in approximate pattern matching, we are given two strings P and S and an *error threshold* k. The goal is then to find all ending positions of substrings of S that are similar to P up to k errors. There are various ways to define k errors. Perhaps the most popular one is the *edit distance* metric, where k is a bound on the number of insertions, deletions, and substitutions needed to convert one substring to the other.

In classical (uncompressed) approximate pattern matching, a simple dynamic programming solution of Sellers [45] solves this problem (under edit distance) in $O(Nm)$ time and $O(m)$ space, where N and m are the lengths of S and P respectively. Several improvements of this result are known, see e.g., the survey by Navarro [39]. Two well known improvements for small values of k are the $O(Nk)$ time algorithm of Landau and Vishkin [33] and the $O(Nk^4/m + N)$ time algorithm of Cole and Hariharan [18]. Both of these can be implemented in $O(m)$ space. The use of compression led to many speedups using various compression schemes [3, 9, 12–14, 19, 27–29, 36, 38]. Many of these speedups are only for checking whether P is similar to S (rather than asking about all substrings of S). The most closely related to our work is approximate pattern matching for LZ78 and LZW compressed strings [12,28,38], which can be solved in time $O(n(\min\{mk, k^4 + m\}) + \text{occ})$ [12]. Here, n corresponds to the compressed length under the LZ compression.

Theorem 3 gives us the first non-trivial algorithms for approximate pattern matching over any grammar compressed string. For instance, if we plug in the Landau-Viskin or Cole-Hariharan algorithms in Theorem 3(ii) we obtain an algorithm of $O(n(\min\{mk, k^4 + m\} + \log N) + \text{occ})$ time and $O(n \cdot \alpha_k(n) + m + \text{occ})$ space. It is important to note that any algorithm (not only Landau-Viskin or Cole-Hariharan) and any similarity measure (not only edit distance) can be applied to Theorem 3. For example, under the Hamming distance measure we can combine our algorithm with a fast algorithm for the (uncompressed) approximate string matching problem for the Hamming distance measure [5].

Outline and Techniques

Before diving into the details, we give an outline of the paper and of the new techniques and data structures that we introduce and believe to be of independent interest.

Let $\mathcal{S}$ be a SLP of size n representing a string of length N. We begin in Section 2 by defining a forest H of size n that represents the heavy paths [26] in the parse tree of $\mathcal{S}$. We then combine the forest H with an existing *weighted ancestor* data structure[2], leading to a solution to the random access problem in the bounds of Theorem 1(i).

The main part of the paper focuses on reducing the random access time to $O(\log N)$. This is achieved by designing a new weighted ancestor data structure. In Section 3 we describe the building block of this data structure – the *interval-biased search tree*. An interval-biased search tree is a new and simple linear time constructible predecessor data structure. The query complexity of predecessor(p) on this data structure is proportional to the integer |successor(p) – predecessor(p)|. This is designed in such a way that if we assign one interval-biased search tree for every root-to-leaf path in the representation H then a weighted ancestor query on H translates to $O(\log N)$ predecessor queries whose total time sums up to $O(\log N)$. However, this is at the cost of $O(n^2)$ preprocessing time and space as there can be $O(n)$ root-to-leaf paths each assigned to an $O(n)$-sized interval-biased search tree. The goal of Section 4 is then to reduce this quadratic preprocessing to be only an inverse Ackermann factor away from linear.

To achieve this, we utilize the overlaps between root-to-leaf paths in H as captured by another heavy-path decomposition, this time of H itself. Assigning one interval-biased search tree for every disjoint heavy path in this decomposition requires only $O(n)$ total preprocessing. We are then left with the problem of navigating between these paths during a random access query. We show that this translates to a weighted ancestor query on a related tree L. The tree L has a vertex for each path in the decomposition of H and an edge for each adjacent paths. While the length of a root-to-leaf path in H can be arbitrary, in L it is always bounded by $O(\log n)$. This fact is enough to reduce the preprocessing to $O(n \log n)$.

To further reduce the preprocessing, we partition L into disjoint trees in the spirit of Alstrup, Husfeldt, and Rauhe's decomposition [2] for solving the *marked ancestor* problem. One of these trees has $O(n/\log n)$ leaves and so we can apply the solution above for $O(n)$ preprocessing. The other trees all have $O(\log n)$ leaves and we handle them recursively. However, before we can recurse on these trees they are modified so that each has $O(\log n)$ vertices (rather than leaves). This is done by another type of path decomposition (i.e. not a heavy-path decomposition) of L. By carefully choosing the desired sizes of the recursion subproblems we get the bounds of Theorem 1(ii).

We extend both random access solutions to the substring decompression problem in Section 5. To do this efficiently we augment our data structures with a linear number of pointers. These pointers allow us to compute the roots of the subtrees of the parse tree that must be expanded to produce the desired substring using only two random access computations. The subsequent expansion of the substring can be done in linear time in the length of the substring, leading to the bounds of Theorem 2.

Finally, in Section 6 we combine our substring decompression result with a technique for compressed approximate string matching on LZ78 and LZW compressed string [12] to obtain an algorithm for grammar compressed strings. The algorithm computes the approximate occurrences of the pattern in a single bottom-up traversal of the grammar. At each step we use the substring decompression algorithm to decode a relevant small portion of string thereby avoiding a full decompression.

[2] A weighted ancestor query (v, p) asks for the lowest ancestor of v whose weighted distance from v is at least p.

2 Fast Random Access in Linear-Space

In the rest of the paper, we let $\mathcal{S}$ denote an SLP of size n representing a string of length N, and let T be the corresponding parse tree (see Fig. 1(b)). Below we present an $O(n)$ space representation of $\mathcal{S}$ that supports random access in $O(\log N \log\log N)$ time, leading to Theorem 1(i). To achieve this we partition $\mathcal{S}$ into disjoint paths according to a *heavy path decomposition* [26] of the parse tree T and combine this with fast predecessor data structures to search in these heavy paths. This achieves the desired time bound but uses $O(n^2)$ preprocessing time and space. We introduce a simple but compact representation of all these predecessor data structures that reduces the space to $O(n)$.

2.1 Heavy Path Decompositions

Similar to Harel and Tarjan [26], we define the *heavy path decomposition* of the parse tree T as follows. For each node v define $T(v)$ to be the subtree rooted at v and let size(v) be the number of descendant leaves of v. We classify each node in T as either *heavy* or *light*. The root is light. For each internal node v we pick a child of maximum size and classify it as heavy. The heavy child of v is denoted heavy(v). The remaining children are light. An edge to a light child is a *light edge* and an edge to a heavy child is a *heavy edge*. Removing the light edges we partition T into *heavy paths*. A *heavy path suffix* is a simple path $v_1, \ldots, v_k$ from a node v_1 to a leaf in $T(v_1)$, such that $v_{i+1} = \text{heavy}(v_i)$, for $i = 1, \ldots, k-1$.

If u is a light child of v then size$(u) \leq$ size$(v)/2$ since otherwise u would be heavy. Consequently, the number of light edges on a path from the root to a leaf is at most $O(\log N)$ [26]. Note that our definition of heavy paths is slightly different than the usual one. We construct our heavy paths according to the number of leaves of the subtrees and not the total number nodes.

We extend heavy path decomposition of trees to SLPs in a straightforward manner. We consider each grammar variable v as a node in the directed acyclic graph defined by the grammar (see Fig. 1(c)). For a node v in $\mathcal{S}$ let $S(v)$ be the substring induced by the parse tree rooted at v and define the size of v to be the length of $S(v)$. We define the heavy paths in $\mathcal{S}$ as in T from the size of each node. Since the size of a node v in $\mathcal{S}$ is the number of leaves in $T(v)$ the heavy paths are well-defined and we may reuse all of the terminology for trees on SLPs. In a single $O(n)$ time bottom-up traversal of $\mathcal{S}$ we can compute the sizes of all nodes and hence the heavy path decomposition of $\mathcal{S}$.

2.2 Fast Random Access

We first give an $O(\log N \log\log N)$ time and $O(n^2)$ preprocessing time and space solution. Our data structure represents the following information for each heavy path suffix $v_1, \ldots, v_k$ in $\mathcal{S}$.

- The length size(v_1) of the string $S(v_1)$.
- The index z of v_k in the left-to-right order of the leaves in $T(v_1)$ and the character $S(v_1)[z]$.
- A predecessor data structure for the *left size sequence* $l_0, l_1, \ldots, l_k$, where l_i is the sum of 1 plus the sizes of the left and light children of the first i nodes in the heavy path suffix.
- A predecessor data structure for the *right size sequence* $r_0, \ldots, r_k$, where r_i is the sum of 1 plus the sizes of the right and light children of the first i nodes in the heavy path suffix.

With the above information we can perform a top down search of T as follows. Suppose that we have reached node v_1 with heavy path suffix $v_1, \ldots, v_k$ and our goal is to access the character $S(v_1)[p]$. We then compare p with the index z of v_k. There are three cases:

1. If $p = z$ we report the stored character $S(v_1)[z]$ and end the search.
2. If $p < z$ we compute the predecessor l_i of p in the left size sequence. We continue the top down search from the left child u of v_{i+1}. The position of p in $T(u)$ is $p - l_i + 1$.

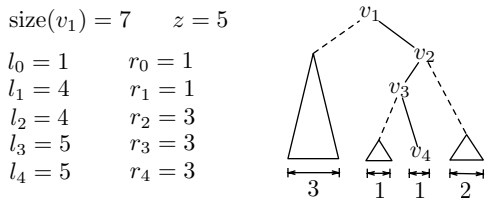


Figure 2: The left and right size sequences for a heavy path suffix v_1, v_2, v_3, v_4. The dotted edges are to light subtrees and the numbers in the bottom are subtree sizes. A search for $p = 5$ returns the stored character for $S(v_1)[z]$. A search for $p = 4$ computes the predecessor l_2 of 4 in the left size sequence. The search continues in the left subtree of v_3 for position $p - l_2 + 1 = 4 - 4 + 1 = 1$. A search for $p = 6$ computes the predecessor r_1 of $7 - 6 = 1$ in the right size sequence. The search continues in the right subtree of v_2 for position $p - z = 6 - 5 = 1$.

3. If $p > z$ we compute the predecessor r_i of $\text{size}(v_1) - p$ in the right size sequence. We continue the top down search from the right child u of v_{i+1}. The position of p in $T(u)$ is $p - (z + \sum_{j=i+2}^{k} \text{size}(v_j))$ (note that we can compute the sum in constant time as $r_k - r_{i+2}$).

An example of the representation and the search algorithm is shown in Fig. 2. The complexity of the algorithm depends on the implementation of the predecessor data structures. With a van Emde Boas data structure [50] we can answer predecessor queries in $O(\log \log N)$ time. Our algorithm visits at most $O(\log N)$ heavy paths during a top down traversal of the parse tree and therefore we use $O(\log N \log \log N)$ time to do a random access. However, the total length of all heavy path suffixes is $\Omega(n^2)$ (in the worst-case). Hence, even with a linear space implementation of the van Emde Boas data structures [52] we use $\Omega(n^2)$ space in total.

2.3 A Linear Space Representation of Heavy Path Suffixes

We show how to compactly represent all of the predecessor data structures from the algorithm of the previous section in $O(n)$ space.

We introduce the *heavy path suffix forest* H of $\mathcal{S}$. The nodes of H are the nodes of S and a node u is the parent of v in H iff u is the heavy child of v in $\mathcal{S}$. Thus, a heavy path suffix $v_1, \ldots, v_k$ in $\mathcal{S}$ is a sequence of ancestors from v_1 in H. We label the edge from v to its parent u by a left weight and right weight defined as follows. If u is the left child of v in $\mathcal{S}$ the left weight is 0 and the right weight is $\text{size}(v')$ where v' is the right child of v. Otherwise, the right weight is 0 and the left weight is $\text{size}(v')$ where v' is the left child of v. Heavy path suffixes in $\mathcal{S}$ consist of unique nodes and therefore H is a forest. A heavy path suffix in $\mathcal{S}$ ends at one of $|\Sigma|$ leaves in $\mathcal{S}$ and therefore H consists of $|\Sigma|$ trees each rooted at a unique character of Σ. The total size of H is $O(n)$ and we may easily compute it from the heavy path decomposition of $\mathcal{S}$ in $O(n)$ time.

A predecessor query on a left size sequence and right size sequence of a heavy path suffix $v_1, \ldots, v_k$ is now equivalent to a *weighted ancestor query* on the left weights and right weights of H, respectively. Farach-Colton and Muthukrishnan [21] showed how to support weighted ancestor queries in $O(\log \log N)$ time after $O(n)$ space and preprocessing time. Hence, if we plug this in to our algorithm we obtain $O(\log N \log \log N)$ query time with $O(n)$ preprocessing time and space. In summary, we have the following result.

Theorem 4 *For an SLP $\mathcal{S}$ of size n representing a string of length N we can support random access in time $O(\log N \log \log N)$ after $O(n)$ preprocessing time and space.*

3 Interval-Biased Search Trees

We have so far seen how to obtain $O(\log N \log\log N)$ random access time on an SLP $\mathcal{S}$. Our goal now is to reduce this to $O(\log N)$. Recall that $O(\log N \log\log N)$ was a result of preforming $O(\log N)$ predecessor(p) queries, each in $O(\log\log N)$ time. In this section, we introduce a new predecessor data structure – the *interval-biased search tree*. Each predecessor(p) query on this data structure requires $O(\log \frac{N}{x})$ time, where x = successor(p) – predecessor(p).

To see the advantage of $O(\log \frac{N}{x})$ predecessor queries over $O(\log\log N)$, suppose that after performing the predecessor query on the first heavy path of T we discover that the next heavy path to search is the heavy path suffix originating in node u. This means that the first predecessor query takes $O(\log \frac{N}{|S(u)|})$ time. Furthermore, the elements in u's left size sequence (or right size sequence) are all from a universe $\{0, 1, \ldots, |S(u)|\}$. Therefore, the second predecessor query takes $O(\log \frac{|S(u)|}{x})$ where $x = |S(u')|$ for some node u' in $T(u)$. The first two predecessor queries thus require time $O(\log \frac{N}{|S(u)|} + \log \frac{|S(u)|}{x}) = O(\log \frac{N}{x})$. The time required for all $O(\log N)$ predecessor queries telescopes similarly for a total of $O(\log N)$.

We next show how to construct an interval-biased search tree in linear time and space. Simply using this tree as the predecessor data structure for each heavy path suffix of $\mathcal{S}$ already results in the following lemma.

Lemma 1 *For an SLP $\mathcal{S}$ of size n representing a string of length N we can support random access in time $O(\log N)$ after $O(n^2)$ preprocessing time and space.*

3.1 A Description of the Tree

We now define the interval-biased search tree associated with $\hat{n}$ integers $l_1 \leq \ldots \leq l_{\hat{n}}$ from a universe $\{0, 1, \ldots, \hat{N}\}$. For simplicity, we add the elements $l_0 = 0$ and $l_{\hat{n}+1} = \hat{N}$. The interval-biased search tree is a binary tree that stores the intervals $[l_0, l_1], [l_1, l_2], \ldots, [l_{\hat{n}}, l_{\hat{n}+1}]$ with a single interval in each node. We describe the tree recursively as follows:

1. Let i be such that $l_{i+1} - l_i > (l_{\hat{n}+1} - l_0)/2$ (there is at most one such i).
2. If no such i exists then let i be such that $l_i - l_0 \leq (l_{\hat{n}+1} - l_0)/2$ and $l_{\hat{n}+1} - l_{i+1} \leq (l_{\hat{n}+1} - l_0)/2$.
3. The root of the tree stores the interval $[l_i, l_{i+1}]$.
4. The left child of the root is the interval-biased search tree storing the intervals $[l_0, l_1], [l_1, l_2], \ldots, [l_{i-1}, l_i]$.
5. The right is the interval-biased search tree storing the intervals $[l_{i+1}, l_{i+2}], [l_{i+2}, l_{i+3}], \ldots, [l_{\hat{n}}, l_{\hat{n}+1}]$.

When we search the tree for a query p and reach a node corresponding to the interval $[l_i, l_{i+1}]$, we compare p with l_i and l_{i+1}. If $l_i \leq p \leq l_{i+1}$ then we return l_i as the predecessor. If $p < l_i$ (resp. $p > l_{i+1}$) we continue the search in the left child (resp. right child). Notice that an interval $[l_i, l_{i+1}]$ of length $x = l_{i+1} - l_i$ such that $\hat{N}/2^{j-1} \leq x \leq \hat{N}/2^j$ is stored in a node of depth at most j. Therefore, a query p whose predecessor is l_i (and whose successor is l_{i+1}) terminates at a node of depth at most j. The query time is thus $j \leq 1 + \log \frac{\hat{N}}{x} = O(\log \frac{\hat{N}}{x})$ which is exactly what we desire as x = successor(p) – predecessor(p).

3.2 A Linear-Time Construction

We still need to describe an efficient $O(\hat{n})$ time and space top-down construction of the interval-biased search tree. Before we construct the tree, we initialize a Range Maximum Query (RMQ) data structure over the array of interval lengths $(l_1 - l_0), (l_2 - l_1), \ldots, (l_{\hat{n}+1} - l_{\hat{n}})$. Such a data structure can be constructed in $O(\hat{n})$ time and answers RMQ(j,k) queries in constant time [10, 11, 22, 26, 44]. Here, RMQ(j,k) returns $\arg\max_{i=j}^{k}\{l_{i+1} - l_i\}$.

We now show how to construct the interval-biased search tree storing the intervals $[l_j, l_{j+1}], \ldots, [l_k, l_{k+1}]$. We focus on finding the interval $[l_i, l_{i+1}]$ to be stored in its root. The rest of the tree is constructed recursively so that the left child is a tree storing the intervals $[l_j, l_{j+1}], \ldots, [l_{i-1}, l_i]$ and the right child is a tree storing

the intervals $[l_{i+1}, l_{i+2}], \ldots, [l_k, l_{k+1}]$. To identify the interval $[l_i, l_{i+1}]$, we first query RMQ(j,k). If the query returns an interval of length greater than $(l_{k+1} - l_j)/2$ then this interval is chosen as $[l_i, l_{i+1}]$.

Otherwise, we are looking for an interval $[l_i, l_{i+1}]$ such that i is the largest value where $l_i \leq (l_{k+1} + l_j)/2$ holds. We can find this interval in $O(\log(k-j))$ time by doing a binary search for $(l_{k+1} + l_j)/2$ in the subarray $l_j, l_{j+1}, \ldots, l_{k+1}$. However, notice that we are not guaranteed that $[l_i, l_{i+1}]$ partitions the intervals in the middle. In other words, $i - j$ can be much larger than $k - i$ and vice versa. This means that the total time complexity of all the binary searches we do while constructing the entire tree can amount to $O(n \log n)$ and we want $O(n)$. To overcome this, notice that we can find $[l_i, l_{i+1}]$ in $\min\{\log(i-j), \log(k-i)\}$ time if we use a doubling search from both sides of the subarray. That is, if prior to the binary search, we narrow the search space by doing a parallel scan of the elements $l_j, l_{j+2}, l_{j+4}, l_{j+8}, \ldots$ and $l_k, l_{k-2}, l_{k-4}, l_{k-8}, \ldots$. This turns out to be crucial for achieving $O(n)$ total construction time as we now show.

To verify the total construction time, first notice that the range maximum queries require only constant time. We therefore need to compute the total time required for all the binary searches. Let $T(\hat{n})$ denote the time complexity of all the binary searches, then $T(\hat{n}) = T(i) + T(\hat{n} - i) + \min\{\log i, \log(\hat{n} - i)\}$ for some i. Setting $d = \min\{i, \hat{n} - i\} \leq \hat{n}/2$ we get that $T(\hat{n}) = T(d) + T(\hat{n} - d) + \log d$ for some $d \leq \hat{n}/2$, which is equal[3] to $O(\hat{n})$.

3.3 Final Tuning

Our data structure so far requires $O(\log \frac{\hat{N}}{x})$ time to answer a predecessor(p) query where x = successor(p) – predecessor(p). We have seen that using this data structure for each heavy path suffix of $\mathcal{S}$ can reduce the random access time to $O(\log N)$. However, this is currently at the cost of $O(n^2)$ preprocessing time and space. In the next section, we reduce the preprocessing to linear. This will be achieved by utilizing the overlaps between the domains of these n data structures.

For that, we actually need one last important property[4] from the interval-biased search tree. Suppose that right before doing a predecessor(p) query we know that $p > l_k$ for some k. We can make sure that the query time is reduced to $O(\log \frac{\hat{N} - l_k}{x})$. To achieve this, in a single traversal of the tree, we compute for each node its lowest common ancestor with the node $[l_{\hat{n}}, l_{\hat{n}+1}]$. Then, when searching for p, we can start the search in the lowest common ancestor of $[l_k, l_{k+1}]$ and $[l_{\hat{n}}, l_{\hat{n}+1}]$ in the interval-biased search tree.

4 Closing the Time-Space Tradeoffs for Random Access

We have so far seen two ideas that together offer a time-space tradeoff for random accessing an SLP. The first idea was to use an interval-biased search tree as a predecessor data structure. Storing such a data structure for each of the n heavy path suffixes yields $O(\log N)$ random access time after $O(n^2)$ preprocessing. The second idea was to utilize the overlaps between the domains of these n data structures. These overlaps are captured by the *heavy path suffix forest* H whose size is only $O(n)$. Using a *weighted ancestor* data structure on H the preprocessing is reduced to $O(n)$ at the cost of $O(\log N \log \log N)$ random access time.

In this section we aim at closing this gap between $O(n)$ preprocessing and $O(\log N)$ random access time. This will be achieved by designing a novel *weighted ancestor* data structure on H whose building blocks include interval-biased search trees. To do so, we actually perform a heavy path decomposition of H itself. For each heavy path P in this decomposition we keep one interval-biased search tree for its left size sequence and another one for its right size sequence. It is easy to see that the total size of all these interval-biased search trees is bounded by $O(n)$. We focus on queries of the left size sequence, the right size sequence is handled similarly.

Let P be a heavy path in the decomposition, let v be a vertex on this path, and let $w(v, v')$ be the weight of the edge between v and his child v', We denote by $b(v)$ the weight of the part of P below v and

[3] By an inductive assumption that $T(\hat{n}) < 2\hat{n} - \log \hat{n} - 2$ we get that $T(\hat{n})$ is at most $2d - \log d - 2 + 2(\hat{n} - d) - \log(\hat{n} - d) - 2 + \log d = 2\hat{n} - \log(\hat{n} - d) - 4$, which is at most $2\hat{n} - \log \hat{n} - 3$ since $d \leq \hat{n}/2$.

[4] In fact, there exist linear-time-constructable predecessor data structures with query complexity only $O(\log \log \frac{\hat{N}}{x})$ [42]. They are more complicated than our tree, but more importantly, their query time cannot handle $\hat{N}$ reducing to $\hat{N} - l_k$.

by $t(v)$ the weight above v. As an example, consider the green heavy path $P = (v_5\text{-}v_4\text{-}v_8\text{-}v_9)$ in Fig. 3, then $b(v_4) = w(v_4, v_8) + w(v_8, v_9)$ and $t(v_4) = w(v_5, v_4)$. In general, if $P = (v_k\text{-}v_{k-1}\text{-}\cdots\text{-}v_1)$ then v_1 is a leaf in H and $b(v_{i+1})$ is the i'th element in P's predecessor data structure. The $b(\cdot)$ and $t(\cdot)$ values of all vertices can easily be computed in $O(n)$ time.

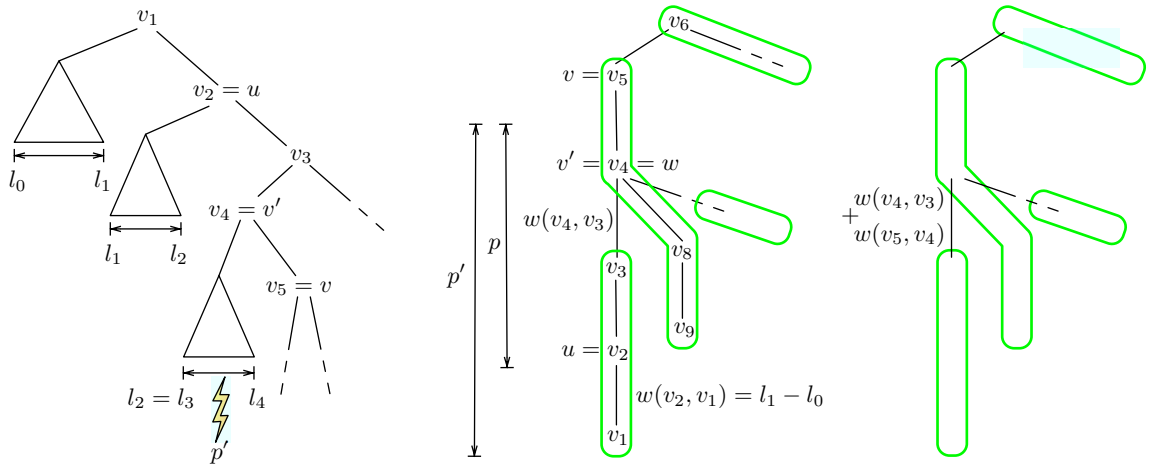


Figure 3: The parse tree T of an SLP is depicted on the left, the *heavy path suffix forest* H is in the middle, and the *light representation* L of H is on the right. The heavy path decomposition of H is marked (in green) and defines the vertex set of L.

Recall that given any vertex u in H and any $0 \leq p \leq N$ we need to be able to find the lowest ancestor v of u whose weighted distance from u is at least p. If we want the total random access time to be $O(\log N)$ then finding v should be done in $O\big(\log \frac{|S(u)|}{w(v,v')}\big)$ time where v' is the child of v which is also an ancestor of u. Suppose that both u and v are on the same heavy path P in the decomposition. In this case, a single predecessor(p') query on P would indeed find v in $O(\log \frac{t(u)}{w(v,v')}) = O\big(\log \frac{|S(u)|}{w(v,v')}\big)$ time, here $p' = p + b(u)$. This follows from the property we described in Section 3.3.

The problem is thus to locate v when, in the decomposition of H, v is on the heavy path P but u is not. To do so, we first locate a vertex w that is both an ancestor of u and belongs to P. Once w is found, if its weighted distance from u is greater than p then $v = w$. Otherwise, a single predecessor(p'') query on P finds v in $O(\log \frac{t(w)}{w(v,v')})$ time, which is $O\big(\log \frac{|S(u)|}{w(v,v')}\big)$ since $t(w) \leq |S(u)|$. Here, $p'' = p$ - weight(path from u to w in H) + $b(w)$. We are therefore only left with the problem of finding w and the weight of the path from u to w.

4.1 A Light Representation of Heavy paths

In order to navigate from u up to w we introduce the *light representation* L of H. Intuitively, L is a (non-binary) tree that captures the light edges in the heavy-path decomposition of H. Every path P in the decomposition of H corresponds to a single vertex P in L, and every light edge in the decomposition of H corresponds to an edge in L. If a light edge e in H connects a vertex w with its child then the weight of the corresponding edge in L is the original weight of e plus $t(w)$. (See the edge of weight $w(v_4, v_3) + w(v_5, v_4)$ in Fig. 3).

The problem of locating w in H now translates to a weighted ancestor query on L. Indeed, if u belongs to a heavy-path P' then P' is also a vertex in L and locating w translates to finding the lowest ancestor of P' in L whose weighted distance from P' is at least $p - t(u)$. However, a weighted ancestor data structure on L would be too costly. Instead, we utilize the important fact that the height of L is only $O(\log n)$. This

follows from the edges of L corresponding to only light edges of H.

We can therefore construct, for every root-to-leaf path in L, an interval-biased search tree as its predecessor data structure. The total time and space for constructing these data structures is $O(n \log n)$. A query for finding the ancestor of P' in L whose weighted distance from P' is at least $p - t(u)$ can then be done in $O(\log \frac{|S(u)|}{t(w)})$ time. This is $O\big(\log \frac{|S(u)|}{w(v,v')}\big)$ as $w(v, v') \leq t(w)$. We summarize this with the following lemma.

Lemma 2 *For an SLP $\mathcal{S}$ of size n representing a string of length N we can support random access in time $O(\log N)$ after $O(n \log n)$ preprocessing time and space.*

4.2 An Inverse-Ackerman Type bound

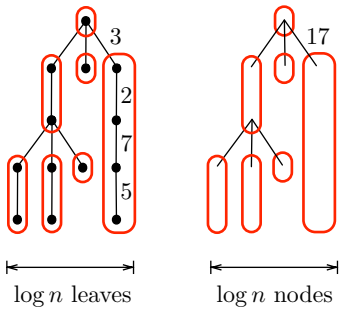
We have just seen that after $O(n \log n)$ preprocessing we can support random access in $O(\log N)$ time. This superlinear preprocessing originates in the $O(n \log n)$-sized data structure that we construct on L for $O\big(\log \frac{|S(u)|}{w(v,v')}\big)$-time weighted ancestor queries. We now turn to reducing the preprocessing to be arbitrarily close to linear by recursively shrinking the size of this weighted ancestor data structure on L.

In order to do so, we perform a decomposition of L that was originally introduced by Alstrup, Husfeldt, and Rauhe [2] for solving the the *marked ancestor* problem: Given the rooted tree L of n nodes, for every maximally high node whose subtree contains no more than $\log n$ *leaves*, we designate the subtree rooted at this node a *bottom tree*. Nodes not in a bottom tree make up the *top tree*. It is easy to show that the top tree has at most $n/\log n$ leaves and that this decomposition can be done in linear time.

Notice that we can afford to construct, for every root-to-leaf path *in the top tree*, an interval-biased search tree as its predecessor data structure. This is because there will be only $n/\log n$ such data structures and each is of size $\text{height}(L) = O(\log n)$. In this way, a weighted ancestor query that originates in a top tree node takes $O\big(\log \frac{|S(u)|}{w(v,v')}\big)$ time as required. The problem is therefore handling queries originating in bottom trees.

To handle such queries, we would like to recursively apply our $O(n \log n)$ weighted ancestor data structure on each one of the bottom trees. This would work nicely if the number of *nodes* in a bottom tree was $O(\log n)$. Unfortunately, we only know this about the number of its *leaves*. We therefore use a *branching representation* B for each bottom tree. The number of *nodes* in the representation B is indeed $\log n$ and it is defined as follows.

We first partition L into disjoint paths according to the following rule: A node v belongs to the same path as its child unless v is a branching-node (has more than one child). We associate each path P in this decomposition with a unique interval-biased search tree as its predecessor's data structure. The *branching representation* B is defined as follows. Every path P corresponds to a single node in B. An edge e connecting path P' with its parent-path P corresponds to an edge in B whose weight is e's original weight plus the total weighted length of the path P' (See Fig. 4).



On the left is some *bottom tree* – a weighted tree with $\log n$ leaves. The bottom tree can be decomposed into $\log n$ paths (marked in red) each with at most one branching node. Replacing each such path with a single node we get the *branching representation* B as depicted on the right. The edge-weight 17 is obtained by the original weight 3 plus the weighted path 2+7+5.

Figure 4: A bottom tree and its branching representation B

Each internal node in B has at least two children and therefore the number of nodes in B is $O(\log n)$. Furthermore, similarly to Section 4.1, our only remaining problem is weighted ancestor queries on B. Once the correct node is found in B, we can query the interval-biased search tree of its corresponding path in L in $O\big(\log \frac{|S(u)|}{w(v,v')}\big)$ time as required.

Now that we can capture a bottom tree with its branching representation B of logarithmic size, we could simply use our $O(n \log n)$ weighted ancestor data structure on every B. This would require an $O(\log n \log\log n)$-time construction for each one of the $n/\log n$ bottom trees for a total of $O(n \log\log n)$ construction time. In addition, every bottom tree node v stores its weighted distance $d(v)$ from the root of its bottom tree. After this preprocessing, upon query v, we first check $d(v)$ to see whether the target node is in the bottom tree or the top tree. Then, a single predecessor query on the (bottom or top) tree takes $O\big(\log \frac{|S(u)|}{w(v,v')}\big)$ time as required.

It follows that we can now support random access on an SLP in time $O(\log N)$ after only $O(n \log\log n)$ preprocessing. In a similar manner we can use this $O(n \log\log n)$ preprocessing recursively on every B to obtain an $O(n \log\log\log n)$ solution. Consequently, we can reduce the preprocessing to $O(n \log^* n)$ while maintaining $O(\log N)$ random access. Notice that if we do this naively then the query time increases by a $\log^* n$ factor due to the $\log^* n$ $d(v)$ values we have to check. To avoid this, we simply use an interval-biased search tree for every root-to-leaf path of $\log^* n$ $d(v)$ values. This only requires an additional $O(n \log^* n)$ preprocessing and the entire query remains $O\big(\log \frac{|S(u)|}{w(v,v')}\big)$.

Finally, we note that choosing the recursive sizes more carefully (in the spirit of [1, 16]) can reduce the $\log^* n$ factor down to $\alpha_k(n)$ for any fixed k. In summary, we have the following result.

Theorem 5 *For an SLP $\mathcal{S}$ of size n representing a string of length N we can support random access in time $O(\log N)$ after $O(n \cdot \alpha_k(n))$ preprocessing time and space for any fixed k.*

Combining Theorems 4 and 5 gives us Theorem 1.

5 Substring Decompression

We now extend our random access solutions from the previous section to efficiently support substring decompression. Note that we can always decompress a substring of length m using m random access computations. In this section we show how to do it using just 2 random access computations and additional $O(m)$ time. This immediately implies Theorem 2.

We the extend the representation of $\mathcal{S}$ as follows. For each node v in $\mathcal{S}$ we add a pointer to the next descendant node on the heavy path suffix for v whose light child is to the left of the heavy path suffix and to the right of the heavy path suffix, respectively. This increases the space of the data structure by only a constant factor. Furthermore, we may compute these pointers during the construction of the heavy path decomposition of $\mathcal{S}$ without increasing the asymptotic complexity.

We decompress a substring $S[i, j]$ of length $m = j - i$ as follows. First, we compute the lowest common ancestor v of the search paths for i and j by doing a top-down search for i and j in parallel. We then continue the search for i and j independently. Along each heavy-path on the search for i we collect all subtrees to the left of the heavy path in a linked list using the above pointers. The concatenation of the linked list is the roots of subtrees to left of the search path from v to i. Similarly, we compute the linked list of subtrees to the right of the search path from v to j. Finally, we decode the subtrees from the linked lists thereby producing the string $S[i, j]$.

With our added pointers we construct the linked lists in time proportional to the length of the lists which is $O(m)$. Decoding each subtree uses time proportional to the size of the subtree. The total sizes of the subtrees is $O(m)$ and therefore decoding also takes $O(m)$ time. Adding the time for the two random access computations for i and j we obtain Theorem 2.

6 Compressed Approximate String Matching

We now show how to efficiently solve the compressed approximate string matching problem for grammar compressed strings. Let P and be string of length m and let k be an error threshold. We assume that the algorithms for the uncompressed problem produces the matches in sorted (as is the case for all solution that we are aware of). Otherwise, additional time for sorting should be included in the bounds.

To find all approximate occurrences of P within S without decompressing $\mathcal{S}$ we combine our substring decompression solution from the previous section with a technique for compressed approximate string matching on LZ78 and LZW compressed string [12].

We find the occurrences of P in S in a single bottom-up traversal of $\mathcal{S}$ using an algorithm for (uncompressed) approximate string matching as a black-box. At each node v in $\mathcal{S}$ we compute the matches of P in $S(v)$. If v is a leaf we decompress the single character string $S(v)$ in constant time and run our approximate string matching algorithm. Otherwise, suppose that v has left child v_l and right child v_r. We have that $S(v) = S(v_l) \cdot S(v_r)$. We decompress the substring S' of $S(v)$ consisting of the $\min\{|S(v_l)|, m+k\}$ last characters of $S(v_l)$ and the $\min\{|S(v_r)|, m+k\}$ first characters of $S(v_r)$ and run our approximate string matching algorithm on P and S'. We compute the set of matches of P in $S(v)$ by merging the list of matches from the matches of P in $S(v_l)$, $S(v_r)$, S' (we assume here that our approximate string matching algorithm produces list of matches in sorted order). This suffices since any approximate match with at most k errors starting in $S(v_l)$ and ending in $S(v_r)$ must be contained within S'.

For each node v in $\mathcal{S}$ we decompress a substring of length $O(m+k) = O(m)$, solve an approximate string matching problem between two strings of length $O(m)$, and merge lists of matches. Since there are n nodes in $\mathcal{S}$ we do n substrings decompression and approximate string matching computations on strings of length m in total. The merging is done on disjoint matches in S and therefore takes $O(\text{occ})$ time, where occ is the total number of matches of P in S. With our substring decompression result from Theorem 2 and an arbitrary approximate string matching algorithm we obtain Theorem 3.

References

[1] N. Alon and B. Schieber. Optimal preprocessing for answering on-line product queries. Technical report, TR-71/87, Institute of Computer Science, Tel Aviv University, 1987.

[2] S. Alstrup, T. Husfeldt, and T. Rauhe. Marked ancestor problems. In *Proceedings of the 39th annual symposium on Foundations Of Computer Science (FOCS)*, pages 534–543, 1998.

[3] A. Amir, G. Benson, and M. Farach. Let sleeping files lie: Pattern matching in Z-compressed files. *Journal of Comp. and Sys. Sciences*, 52(2):299–307, 1996.

[4] A. Amir, G.M. Landau, and D. Sokol. Inplace 2d matching in compressed images. In *Proc. of the 14th annual ACM-SIAM Symposium On Discrete Algorithms, (SODA)*, pages 853–862, 2003.

[5] Amihood Amir, Moshe Lewenstein, and Ely Porat. Faster algorithms for string matching with k mismatches. *J. Algorithms*, 50(2):257–275, 2004. Announced at SODA 2000.

[6] A. Apostolico and S. Lonardi. Some theory and practice of greedy off-line textual substitution. In *Proc. IEEE Data compression conference*, pages 119–128, 1998.

[7] A. Apostolico and S. Lonardi. Compression of biological sequences by greedy off-line textual substitution. In *Proc. IEEE Data compression conference*, pages 143–152, 2000.

[8] A. Apostolico and S. Lonardi. Off-line compression by greedy textual substitution. *Proc. IEEE*, 88(11):1733–1744, 2000.

[9] O. Arbell, G. M. Landau, and J. Mitchell. Edit distance of run-length encoded strings. *Information Processing Letters*, 83(6):307–314, 2001.

[10] M.A. Bender, M. Farach-Colton, G. Pemmasani, S. Skiena, and P. Sumazin. Lowest common ancestors in trees and directed acyclic graphs. *Journal of Algorithms*, 57(2):75–94, 2005.

[11] O. Berkman and U. Vishkin. Recursive star-tree parallel data structure. *SIAM Journal on Computing*, 22(2):221–242, 1993.

[12] Philip Bille, Rolf Fagerberg, and Inge Li Gørtz. Improved approximate string matching and regular expression matching on ziv-lempel compressed texts. *ACM Transactions on Algorithms*. To appear. Announced at CPM 2007.

[13] H. Bunke and J. Csirik. An improved algorithm for computing the edit distance of run length coded strings. *Information Processing Letters*, 54:93–96, 1995.

[14] P. Cégielski, I. Guessarian, Y. Lifshits, and Y. Matiyasevich. Window subsequence problems for compressed texts. In *Proc. of the 1st symp. on Computer Science in Russia (CSR)*, pages 127–136, 2006.

[15] Moses Charikar, Eric Lehman, Ding Liu, Rina Panigrahy, Manoj Prabhakaran, Amit Sahai, and Abhi Shelat. The smallest grammar problem. *IEEE Transactions on Information Theory*, 51(7):2554–2576, 2005. Announced at STOC 2002 and SODA 2002.

[16] B. Chazelle and B. Rosenberg. Computing partial sums in multidimensional arrays. In *Proceedings of the 5th annual ACM Symposium on Computational Geometry (SCG)*, pages 131–139, 1989.

[17] Francisco Claude and Gonzalo Navarro. Self-indexed text compression using straight-line programs. In *Proc. 34th Mathematical Foundations of Computer Science*, volume 5734 of *Lecture Notes in Computer Science*, pages 235–246, 2009.

[18] Richard Cole and Ramesh Hariharan. Approximate string matching: A simpler faster algorithm. *SIAM J. Comput.*, 31(6):1761–1782, 2002.

[19] M. Crochemore, G.M. Landau, and M. Ziv-Ukelson. A subquadratic sequence alignment algorithm for unrestricted scoring matrices. *SIAM Journal on Computing*, 32:1654–1673, 2003.

[20] C. G. de Marcken. *Unsupervised language acquisition*. PhD thesis, MIT, 1996.

[21] Martin Farach and S. Muthukrishnan. Perfect hashing for strings: Formalization and algorithms. In *Proceedings of the 7th Symposium on Combinatorial Pattern Matching (CPM)*, pages 130–140. Springer, 1996.

[22] J. Fischer and V. Heun. Theoretical and practical improvements on the RMQ-problem, with applications to LCA and LCE. In *Proceedings of the 17th symposium on Combinatorial Pattern Matching (CPM)*, pages 36–48, 2006.

[23] P. Gage. A new algorithm for data compression. *The C Users J.*, 12(2):23 – 38, 1994.

[24] Leszek Gasieniec, Roman Kolpakov, Igor Potapov, and Paul Sant. Real-time traversal in grammar-based compressed files. In *Proceedings of the Data Compression Conference*, pages 458–458, 2005.

[25] Leszek Gasieniec and Igor Potapov. Time/space efficient compressed pattern matching. *Fundam. Inf.*, 56(1,2):137–154, 2003.

[26] D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 13(2):338–355, 1984.

[27] D. Hermelin, S. Landau, G.M. Landau, , and O. Weimann. A unified algorithm for accelerating edit-distance via text-compression. In *Proc. of the 26th International Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 529–540, 2009.

[28] J. Karkkainen, G. Navarro, and E. Ukkonen. Approximate string matching over Ziv-Lempel compressed text. In *Proc. of the 11th symposium on Combinatorial Pattern Matching (CPM)*, pages 195–209, 2000.

[29] J. Karkkainen and E. Ukkonen. Lempel-Ziv parsing and sublinear-size index structures for string matching. In *Proc. of the 3rd South American Workshop on String Processing (WSP)*, pages 141–155, 1996.

[30] J. C. Kieffer and E. H. Yang. Grammar based codes: A new class of universal lossless source codes. *IEEE Trans. Inf. Theory*, 46(3):737–754, 2000.

[31] J. C. Kieffer, E. H. Yang, G. J. Nelson, and P. Cosman. Universal lossless compression via multilevel pattern matching. *IEEE Trans. Inf. Theory*, 46(5):1227 – 1245, 2000.

[32] J. K. Lanctot, M. Li, and E. H. Yang. Estimating dna sequence entropy. In *Proc. of the 11th Symposium On Discrete Algorithms*, pages 409–418, 2000.

[33] G. M. Landau and U. Vishkin. Fast parallel and serial approximate string matching. *J. Algorithms*, 10(2):157–169, 1989.

[34] Jesper N. Larsson and Alistair Moffat. Off-line dictionary-based compression. *Proc. IEEE*, 88(11):1722 – 1732, 2000. Announced at DCC 1999.

[35] Y. Lifshits. Processing compressed texts: A tractability border. In *Proc. of the 18th symposium on Combinatorial Pattern Matching (CPM)*, pages 228–240, 2007.

[36] V. Makinen, G. Navarro, and E. Ukkonen. Approximate matching of run-length compressed strings. In *Proc. of the 12th Symposium On Combinatorial Pattern Matching (CPM)*, pages 1–13, 1999.

[37] U. Manber. A text compression scheme that allows fast searching directly in the compressed file. In *Proc of the 5th Symposium On Combinatorial Pattern Matching (CPM)*, pages 31–49, 1994.

[38] G. Navarro, T. Kida, M. Takeda, A. Shinohara, and S. Arikawa. Faster approximate string matching over compressed text. In *Proc. of the 11th Data Compression Conference (DCC)*, pages 459–468, 2001.

[39] Gonzalo Navarro. A guided tour to approximate string matching. *ACM Comput. Surv.*, 33(1):31–88, 2001.

[40] Craig G. Nevill-Manning. *Inferring sequential structure*. PhD thesis, University of Waikato, 1996.

[41] Craig G. Nevill-Manning and Ian H. Witten. Identifying hierarchical strcture in sequences: A linear-time algorithm. *J. Artif. Intell. Res. (JAIR)*, 7:67–82, 1997.

[42] M. Pătraşcu. private communication. 2009.

[43] W. Rytter. Application of Lempel-Ziv factorization to the approximation of grammar-based compression. *Theoretical Computer Science*, 302(1-3):211–222, 2003.

[44] B. Schieber and U. Vishkin. On finding lowest common ancestors: Simplification and parallelization. *SIAM Journal on Computing*, 17:1253–1262, 1988.

[45] P. Sellers. The theory and computation of evolutionary distances: Pattern recognition. *J. Algorithms*, 1(4):359–373, 1980.

[46] Y. Shibata, T. Kida, S. Fukamachi, M. Takeda, A. Shinohara, T. Shinohara, and S. Arikawa. Byte Pair encoding: A text compression scheme that accelerates pattern matching. *Technical Report DOI-TR-161, Department of Informatics, Kyushu University*, 1999.

[47] Y. Shibata, T. Kida, S. Fukamachi, M. Takeda, A. Shinohara, T. Shinohara, and S. Arikawa. Speeding up pattern matching by text compression. In *Proc. of the 4th Italian Conference Algorithms and Complexity (CIAC)*, pages 306–315, 2000.

[48] James A. Storer. *Data compression: Methods and complexity*. PhD thesis, Princeton University, 1978.

[49] James A. Storer and Thomas G. Szymanski. Data compression via textural substitution. *J. ACM*, 29(4):928–951, 1982.

[50] Peter van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Mathematical Systems Theory*, 10:99–127, 1977. Announced at FOCS 1975.

[51] Terry A. Welch. A technique for high-performance data compression. *IEEE Computer*, 17(6):8–19, 1984.

[52] Dan E. Willard. Log-logarithmic worst-case range queries are possible in space $\theta(n)$. *Inf. Process. Lett.*, 17(2):81–84, 1983.

[53] E. H. Yang and J. C. Kieffer. Efficient universal lossless data compression algorithms based on a greedy sequential grammar transform – part one: Without context models. *IEEE Trans. Inf. Theory*, 46(3):755–754, 2000.

[54] J. Ziv and A. Lempel. On the complexity of finite sequences. *IEEE Transactions on Information Theory*, 22(1):75–81, 1976.

[55] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.